UML 1.4 versus UML 2.0 as languages to describe software architectures

Jorge Enrique Pérez-Martínez[1], Almudena Sierra-Alonso[2]

[1] Universidad Politécnica de Madrid, Departamento de Informática Aplicada
Campus Sur de la U.P.M., 28031 Madrid (Spain)
jeperez@eui.upm.es
[2] Universidad Autónoma de Madrid, Escuela Politécnica Superior
Ctra. de Colmenar, km. 15, 28049 Madrid (Spain)
Almudena.sierra@ii.uam.es

Abstract. UML 1.4 is widely accepted as the standard for representing the various software artifacts generated by a development process. For this reason, there have been attempts to use this language to represent the software architecture of systems as well. Unfortunately, these attempts have ended in representations (boxes and lines) already criticized by the software architecture community. Recently, OMG has published a draft that will constitute the future UML 2.0 specification. In this paper we compare the capacities of UML 1.4 and UML 2.0 to describe software architectures. In particular, we study extensions of both UML versions to describe the static view of the C3 architectural style (a simplification of the C2 style). One of the results of this study is the difficulties found when using the UML 2.0 metamodel to describe the concept of connector in a software architecture.

1 Introduction

UML 1.4 [18] has become the standard for representing the software products obtained in the various activities (like requirement acquisition, requirement analysis, system design, or system deployment) of a software development process. For this reason, it is not surprising that there have been attempts to use UML 1.4 to represent the software architecture of an application. However, the language is not designed to syntactically and semantically represent the elements of software architectures, neither using its constructors as they are defined nor adding stereotypes to them. Some works analyzing this problem are [1][5][7][8][9][10][12][14][16][22][23][24][25][29]. Consequently, the only solution is to make a heavyweight extension to the UML 1.4 metamodel. However, this type of extension requires the modification of the language, which in turn implies that the tools processing it would need to be changed, deviating from the standard. The appearance of UML 2.0 [19][20] in the near future could solve (or at least ease) this problem. As indicated in [3], UML 2.0 promises a significant improvement in the way systems are architected.

In this work we present a set of extensions to the UML 1.4 metamodel and to the UML 2.0 metamodel to describe the static view of the C3 architectural style. This

style is a variation of the C2 style [15]. UML 1.4 has been selected versus UML 1.5 because UML 1.4 is more popular and it is more extended than UML 1.5.

The rest of the paper is organized as follows. In Section 2 we describe basic concepts related to software architecture, trying to establish a conceptual reference framework. In Section 3 we describe how we change the C2 architectural style to obtain the C3 style. In Section 4 we present the extension to the UML 1.4 metamodel to represent the static view of the C3 style. In Section 5 we approach this problem using UML 2.0. Section 6 presents a comparison between the results of the two previous sections. Finally, Section 7 presents conclusions and future lines of research.

2 Software Architecture

Due to the recent appearance of the software architecture discipline, there are still several definitions of this concept. For example, in [2] we find: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." In [28] we can read: "Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns." IEEE Standard 1471 [11] defines architecture as: "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." Our work assumes the definition of software architecture given by [11] since it is the most complete of those referenced.

On the other hand, this work takes the definition of architectural style provided by [4]: "an architectural style is a specialization of a viewtype's elements and relationships, together with a set of constraints on how they can be used. A style defines a family of architectures that satisfy the constraints."

3 The C3 Architectural Style

C3 is an architectural style derived from the C2 style [15]. We briefly describe now the C2 architectural style. "The C2 architectural style can be informally summarized as a network of concurrent components hooked together by message routing devices" [15]. Every component has its own control flow and no assumptions are made about the existence of a shared addressing space. The key elements of the C2 architecture are components and connectors.

Components communicate through asynchronous message passing. There are two types of messages: notifications and requests. Notifications are announcements of changes in the state of the internal object of a component. Requests sent by a component indicate service requests to components on top of it. A notification is always sent downward through a C2 architecture while a request is always sent up.

Both components and connectors must have top and bottom domains. The top domain of a component can only be connected to the bottom domain of a connector and its bottom domain can only be connected to the top domain of a connector.

A connector can be connected to any number of components and/or connectors. Components can only communicate through connectors since direct communication between components is forbidden. Two connectors can only be connected from the bottom of one to the top of the other. Connectors are responsible for routing and, potentially, multicasting messages. A secondary responsibility of connectors is message filtering. Connectors can provide the following policies for filtering and delivery of messages: no filtering, notification filtering, message filtering, prioritized, and message sink

The modifications introduced in the C2 style to obtain C3 are the following:

- There is no predetermined type of inheritance between components.
- Interface operations only allow input arguments.
- The type of component and its internal structure are not predetermined.
- Connectors only support the policies of message filtering and message sink.
- Operations in the interface can define preconditions and postconditions.

4 A Proposal of Heavyweight Extension to UML 1.4 Metamodel to Describe the Static View of the C3 Architectural Style

This section presents our proposal to extend the UML 1.4 metamodel to describe the static view of the C3 style. This proposal has been previously presented in [21]. To extend the metamodel we have followed two rules:

- We do not remove existing metaclasses nor modify their syntax or semantics.
- The new metaclasses must have as few relations as possible with the metaclasses already defined, i.e., they must be self-contained (as much as possible).

The objective behind these rules is to simplify the implementation of this extension in tools that already support the current UML 1.4 metamodel.

We will introduce the new metaclasses to the UML 1.4 metamodel to represent the structural aspects of the C3 architectural style in a new package which we call *C3Description*, located in the package *Foundation*. The abstract syntax for the package *Foundation::C3Description* is shown in Figure 1. In this Figure, the new metaclasses added to the UML 1.4 metamodel appear shading.

Although not shown in that figure, the new constructors (except for *Role* and *Port*) are added to the UML 1.4 metamodel as subclasses of *ModelElement*, which defines the metaattribute *name*. *ModelElement* is a subclass of *Element*, the root metaclass. Constructors *Role* and *Port* will be subclasses of *Element* since they do not need to have a name. As shown in Figure 1, we use the constructors *Attribute*, *Constraint* and *Parameter* defined in package *Core*, and types *Boolean* and *ProcedureExpression* defined in package *Data Types*. A detailed description of the *Foundation::C3Description* package can be found in [21]. To summary we can say:

- The metaclasses *Component*, *Connector* and *Architecture* represent the concepts of component, connector and configuration of the C3 style. The associa-

tion *contains* between *Component/Connector* and *Architecture* indicates that a component and a connector can be composite elements.

- The metaclasses *Port* and *Role* represent the interaction points of a component and a connector, respectively. The cardinality (2) between *Component* and *Port* indicates that a component has two domains: top and bottom. The cardinality (1..*) between *Connector* and *Role* indicates that a connector has several interaction points. The association between *Port* and *Role* models the connection between a component (at one domain) and a connector. The association *conTocon* between *Role* and *Role* models the connection between two different connectors.
- The metaclass *InterfaceElement* models the interface of a C3 component in both domains. An interface element has a name (inherited of *ModelElement*), a direction that indicates whether the component provides this operation to the environment (prov) or whether the component requires that operation from environment (req), a parameters set and, probably, a precondition and a postcondition.
- The state variables of a component are modeled with the metaclass *Attribute* and the invariant of the component with the metaclass *Constraint*.
- Lastly, the metaclass *Filter* models the filter mechanisms supported by C3.

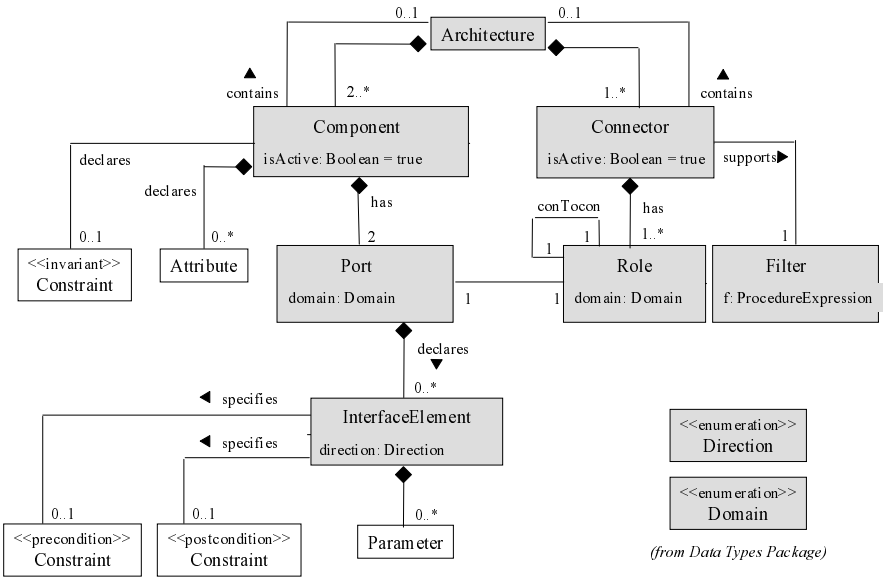

Fig. 1. Abstract syntax of the package Foundation::C3Description in UML 1.4 to describe the static view of the C3 architectural style (from ACM/SIGSOFT 28(3))

5 Description of the Static View of the C3 Architectural Style with UML 2.0

To investigate how to use UML 2.0 to describe the static view of the C3 architectural style we can follow this strategy:

1. Use the elements of UML 2.0, as they are defined by the language.
2. If the previous option is unable to represent the C3 style, the only possibility is extending UML 2.0. We can extend UML 2.0 in two ways:
   - We can define a new dialect of UML 2.0 by using Profiles to customize the language for particular platforms and domains. This implies making use of the package *InfrastrutureLibrary::Profiles*.
   - We can specify a new language related to UML 2.0 by reusing part of the *InfrastructureLibrary::Core* package and augmenting it with appropriate metaclasses and metarelationships. With this approximation we define a new member of the UML 2.0 family of languages.

In the following, we study each of these approximations to evaluate the capabilities of UML 2.0 to describe the static view of the C3 architectural style.

5.1 Using UML 2.0 "as is"

In this case we try to use the constructors defined by the package *UML* to represent the architectural elements of the C3 style. Unfortunately, the constructors defined in UML 2.0 do not allow specifying many of the architectural aspects of the C3 style. Some sample problems are:

1. The semantics of a connector is different in UML 2.0 and in C3. For example, UML 2.0 [20] defines an assembly connector as follows: "is a connector between two components that defines that one component provides the services that another component requires." In C3, a connector can be connected to any number of connectors and not only to components.
2. A component in UML 2.0 (metaclass *Components::Component*) can have any number of associated ports. In C3, each of the two component domains could be modeled by a port, so that a component would only have two associated ports.
3. In C3 an operation in the specification of an interface does not return any result, while the same concept of operation in UML 2.0 (metaclass *Classes::Kernel::Operation*) allows a result to be returned.
4. In UML 2.0 the declaration of an interface (metaclass *Classes::Interfaces::Interface*) can have some attributes (*properties*) associated while interfaces in C3 only allow to declare operations.

We could point out more problems of UML 2.0 to describe the architectural style C3, but one is enough to require a different strategy. In the next section we describe an approximation to describe the static view of the C3 architectural style by defining a new dialect of UML 2.0.

5.2 Defining a Dialect of UML 2.0

With this approximation, the package *InfrastructureLibrary::Profiles* is used to characterize elements of the package *UML* so that they can be used to describe elements of the architectural style C3. Every package referenced in this section is supposed to be contained in the package *UML* except when its name starts with *InfrastructureLibrary*. For every element of style C3, we will describe which element of UML 2.0 looks more appropriate to represent it and the constraints on such element. Those constraints are written in Object Constraint Language, OCL [18].

5.2.1 Component. To characterize a component of C3 we will use the metaclass *Component* of the package *Components* as the base class. The resulting stereotype will be named *C3Component* (see Figure 2).

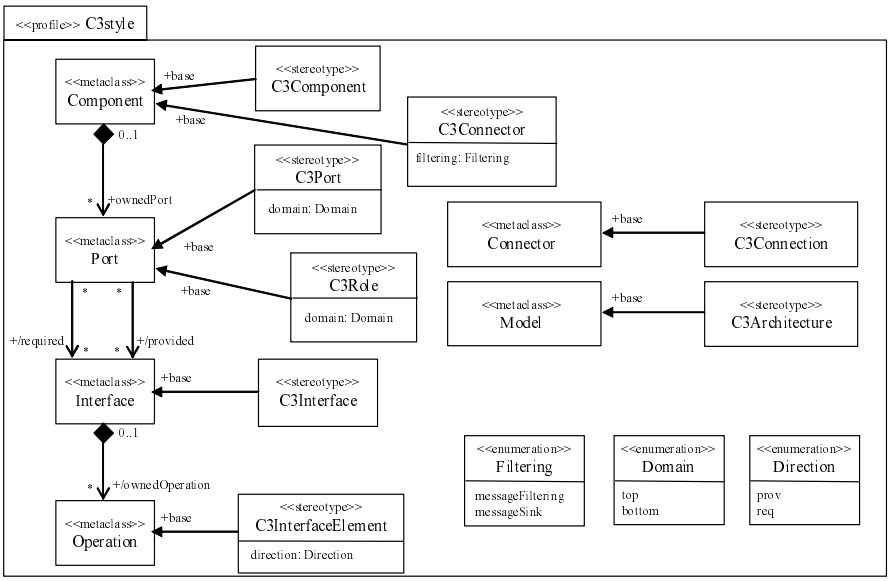

Fig. 2. UML 2.0 profile to describe the static view of the C3 architectural style

We define the following constraints in the context of this stereotype:

1. A *Component* in UML 2.0 is a subclass of *Class* so that it can define attributes and operations. In C3, these features are private of the component.
   self.base.ownedAttribute-> forAll (at| at.visibility = VisibilityKind::private) **and**
   self.base.ownedOperation-> forAll (op| op.visibility = VisibilityKind::private)
2. A C3 component has two ports representing its top and bottom domains. Metaclass *Component* inherits from *Class* and this one inherits from *EncapsuledClassifier,* which declares the association *+ownedPort* (with respect to *Port*).
   self.base.ownedPort -> size() = 2
3. A C3 component can be a simple element or a composite element. This means that it can contain other components and connectors. Metaclass *Component* specifies the relation *+ownedMember,* which extends the concept of basic component to

formalize it as a "building block" with a set of modelling elements. The relation *ownedMember* is defined between metaclasses *Component* and *PackageableElement*. This implies that both component and connector must be *PackageableElement*. In the case of *C3Component* there is no problem since it is a stereotype of *Component* and inherits (indirectly) from *PackageableElement*. However, the case of connector is very different. If we define *C3Connector* as a stereotype of *Connector* (see Section 5.2.5), the inheritance chain of the latter does not go across *PackageableElement*. In fact, *Connector* inherits from *Feature* and this one from *NamedElement* (which inherits from the root metaclass *Element*).

4. In UML 2.0 component interfaces are supported by the component itself (or one of the classifiers implementing it) or are the type of one of its ports. Here we decided to support component interfaces in C3 by the ports of the component.
   self.base.provided -> size() = 0 **and** self.base.required -> size() = 0
5. A component in C3 is an active element. *Component* inherits the attribute *isActive* from *Class* (defined in *CommonBehaviors::Communications*).

5.2.2 InterfaceElement. This constructor represents an operation involved in the interaction of a component with its environment. To characterize an interface element in C3 we take as base class the metaclass *Operation* from the package *Classes::Kernel*. The resulting stereotype will be named *C3InterfaceElement* (see Figure 2). An interface element in C3 defines a direction that indicates if the element represents an operation provided to the environment (*prov*) or an operation required from the environment (*req*). In the context of this stereotype we define the following constraints:

1. An operation declared in the interface does not return any results.
   self.base.returnResult -> size() = 0
2. The parameters from an interface operation of a C3 component can only have *in* direction.
   self.base.parameter -> forAll (p| p.direction = ParameterDirectionKind::in)
3. Only interface operations with direction *prov* can have preconditions and/or post-conditions.
   self.direction = Direction::req **implies**
   self.base.precondition -> empty() **and** self.base.postcondition -> emtpy()

5.2.3 Interface. An interface in C3 is a set of public operations assigned to a component port. To characterize a C3 interface we take as the base class the metaclass *Interface* from the package *Classes::Interfaces*. The resulting stereotype will be named *C3Interface* (see Figure 2). In the context of this stereotype we define the following constraints:

1. A C3 interface does not declare attributes.
   self.base.ownedAttribute -> size() = 0
2. All the operations described in an interface must be of type *C3InterfaceElement*.
   self.base.ownedOperation -> forAll (op|
   stereotype (op).name = 'C3InterfaceElement')
   where: stereotype (c: Class): Stereotype; stereotype = c.extension.ownedEnd.type
3. All the operations defined in the same interface have the same direction

5.2.4 Port. To characterize a port in C3 we will take the metaclass *Port* of the package *CompositeStructures::Ports* as the base class. The resulting stereotype will be called *C3Port* (see Figure 2). A C3 port defines a *domain* corresponding to that of the component it belongs to. In the context of this stereotype, we define the following constraints:

1. The attributes in the metaclass *Port* are constrained by the stereotype as follows. The attribute *isService* is constrained to a value *true* since ports in C3 are external to the component. The attribute *isBehavior* is constrained to have the value *false* since the object or objects inside the component (not the component itself) support this behavior.
2. The interface provided of a port contains only operations with *prov* direction.

```
self.base.provided.ownedOperation -> forAll (op|
                    stereotype(op).direction = Direction::prov)
```

3. The interface required of a port contains only operations with *req* direction.

```
self.base.required.ownedOperation -> forAll (op|
                    stereotype(op).direction = Direction::req)
```

5.2.5 Connector. The description of a C3 connector in UML 2.0 is not as immediate as for the preceding C3 elements. As we pointed out in Section 5.1 the semantics of connectors in UML 2.0 (metaclass *Connector*) does not correspond to the semantics of the same concept in C3. In the following, we describe how some semantic differences between the two constructors can be overcome through the use of constraints. To characterize a C3 connector we will take the metaclass *Connector* from the package *CompositeStructures::InternalStructures* as the base class. The resulting stereotype will be called *C3Connector*. We have to consider the following problems:

1. UML 2.0 [20] defines an assembly connector as follows: "is a connector between two components that defines that one component provides the services that another component requires." In C3, a connector can be connected to any number of connectors, and not only components. In UML 2.0 the end points of a connector must be constructors of the type *ConnectableElement*. UML 2.0 only defines the following metaclasses of this type: *Property*, *Variable*, *Port,* and *Parameter*. Since in UML 2.0 the metaclass *Connector* is not of type *ConnectableElement*, a connector cannot be connected to other connectors. This makes it impossible for *Connector* or any of its stereotypes to represent a C3 connector. The core problem is that the metaclass *Connector* is not a type of *Classifier* as *Component* is. So, connectors in UML 2.0 do not appear to be first class entities.
2. In C3, a connector, like a component, can be formed by components and connectors. However, the metaclass *Connector* is not a *PackageableElement* and consequently it cannot be contained in any component (see Section 5.2.1) nor it can contain other connectors.
3. In C3 a connector has one or more points of interaction with its environment. Each of these points is a role. In UML 2.0, a connector only has two end points, while in C3 a connector can have more than two roles.

Considering the problems we have pointed out, we conclude that the metaclass *Connector* is not valid as the base class for a stereotype representing a connector in

C3. The next obvious option is using the metaclass *Component* as the base class. On the other hand, C3 connectors have a filtering policy associated. So, we add the attribute *filtering* to specify this policy. The resulting stereotype will be named *C3Connector* (see figure 2). In the context of this stereotype we define the following constraints:

1. A C3 connector does not define attributes or operations.
   self.base.ownedAttribute -> size() = 0 **and** self.base.ownedOperation -> size() = 0
2. A C3 connector may have several roles in each domain. If we represent each role with a stereotype of *Port* (see next section), we can describe this constraint as follows:
   self.base.ownedPort -> size() >= 1 **and**
   self.base.ownedPort -> forAll (p| stereotype(p).name = 'C3Role')
3. A C3 connector can be a simple element or a composite element. Specifically, a connector can contain other components or connectors. Observe that by stereotyping the metaclass *Component* to represent a connector we solve the problem pointed out in constraint [3] of section 5.2.1.
4. A C3 connector does not support any interfaces.
   self.base.provided -> size() = 0 **and** self.base.required -> size() = 0
5. A C3 connector defines a filtering policy.
   self.filtering = Filtering::messageFiltering **xor**
   self.filtering = Filtering::messageSink

5.2.6 Role. As we saw in the previous section, we call role to each point of interaction of a connector with its environment. To describe a C3 role in UML 2.0 we find similar problems to those faced to describe a connector. A C3 role represents a point of interaction of a connector with a component (through its port) or with another connector (through its own role). On the contrary, the concept of role in UML 2.0 is defined in the context of a collaboration [20]: "Thus, a collaboration specifies what properties instances must have to be able to participate in the collaboration: A role specifies (through its type) the required set of features a participating instance must have." We can model a role as a stereotype of the metaclass *Port*. The resulting stereotype is named *C3Role* (see Figure 2). In the context of this stereotype we define the following restrictions:

1. A role in C3 does not support any interface.
2. A role associated to a connector in C3 does not support any behavior.
   self.base.isService -> size() = 0 **and** self.base.isBehavior -> size() = 0

5.2.7 Connection Component-Connector and Connector-Connector. In C3, the component port may be linked to the role of a connector. On the other hand, the role of a connector can be linked to the role of another connector or to the port of a component. We have to remember that both *C3Port* and *C3Role* are stereotypes of *Port*. So, how could we state this relationship? It is necessary to define an association between *C3Port* and *C3Role*. However, an association between stereotypes is only possible if it is a subset of the existing associations in the reference metamodel between the base classes of those stereotypes. This means that there must be an association between *Port* and *Port*. Here comes into play the metaclass *Connector*, establishing a link between two instances of type *ConnectableElement* (like instances

of *Port* are). Then, to characterize the connection in C3 between a component port and the role of a connector, or between two roles of two different connectors, we will define a stereotype of the metaclass *Connector* called *C3Connection* (see Figure 2). In the context of this stereotype we define the following constraints:

1. A connection in C3 links two elements.

```
self.base.end -> size() = 2
```

2. A connection in C3 links a component port with a connector role or two roles of two different connectors.

```
let ports: Set = self.base.end.role -> select (el| stereotype(el).name = 'C3Port')
let roles: Set = self base.end.role -> select (el| stereotype(el).name = 'C3Role') in
        ports -> size() = 1 implies roles -> size() = 1 and
        roles -> size() = 2 implies roles -> forAll (r1 r2|
                                        r1.end.connector <> r2.end.connector)
```

3. A connection in C3 cannot link two ports.

```
let ports: Set = self.base.end.role -> select (el| stereotype(el).name = 'C3Port') in
        not ports -> size() = 2
```

5.2.8 Composite Components and Connectors. A C3 architecture is a set of components and connectors whose connectivity respects a set of topological constraints. In Sections 5.2.1 and 5.2.5 we indicated that both a component and a connector may be formed by components and connectors. This means that both a component and a connector may contain an architecture. To characterize a architecture with C3 style we will define a stereotype of *Model* (which is a subclass of *Package*) named *C3Architecture* (see Figure 2). In the context of this stereotype we define the following constraints:

1. A C3 architecture is a composition of components and connectors.
2. A C3 architecture is formed by two or more components and one or more connectors.

5.2.9 An UML 2.0 Profile to Describe the Static View of the C3 Architectural Style. Figure 2 describes the set of stereotypes defined thus far, which have been grouped under the profile *C3style*.

Since style C3 imposes certain topological constraints in relation with the connectivity between components and connectors, it is interesting to show the relationships among the different stereotypes defined. Figure 3 shows those relationships.

5.3 Defining a New Member of the UML 2.0 Language Family

As indicated in the previous section, a connector (represented by the metaclass *Connector*) in UML 2.0 does not appear to be a first class entity. This observation seems to go against the general opinion in the software architecture community, which considers that connectors and components deserve a similar treatment [6][17][26][27]. Moreover, from the document defining the superstructure [20], it seems that a connector in UML 2.0 is limited to connect ports (*ConnectableElement*). The software architecture community assigns more complex semantics to the concept of connector than that represented by the metaclass *Connector*. In this direction, Garlan and Kompanec

[7] indicate that: ”From a run-time perspective, connectors mediate the communication and coordination activities among components. Examples include simple forms of interaction, such as pipes, procedure calls, and event broadcast. Connectors may also represent complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors have interfaces that define the roles played by the participants in the interaction.” In [17], Mehta, Medvidovic and Phadke indicate that: “Connectors can also provide services, such as persistence, invocation, messaging, and transactions, that are largely independent of the interacting components’ functionality.” The same authors [17] say: “Connectors can also have an internal architecture that includes computation and information storage. For example, a load balancing connector would execute an algorithm for switching incoming traffic among a set of components based on knowledge about the current and past load state of components.”

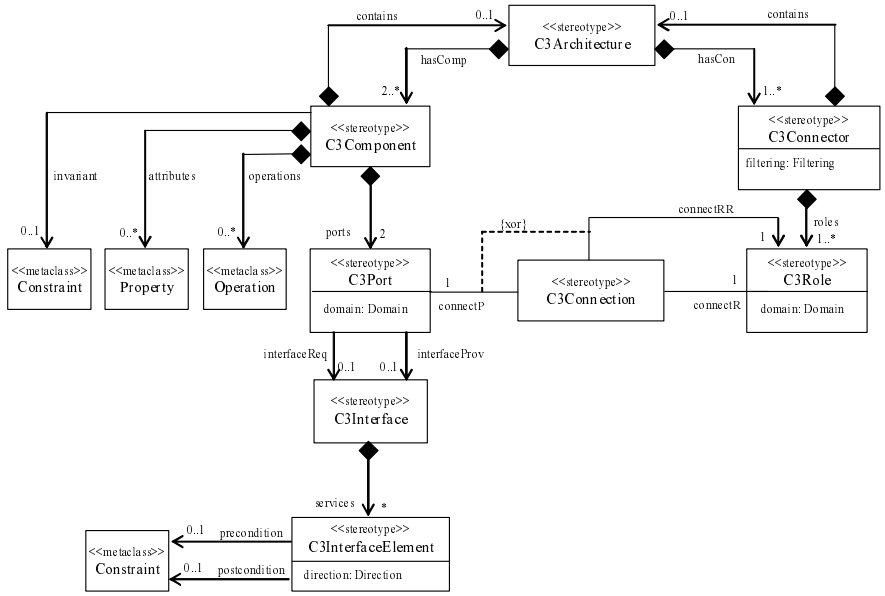

Fig. 3. Abstract syntax of a dialect of UML 2.0 to describe the static view of the C3 architectural style

On the other hand, the concept of connector found in some architectural styles also assigns to it more complex semantics than the one represented by the metaclass *Connector*. For example, in style C3, we have already seen that a connector is responsible for routing messages and implementing the filtering policy defined for it. In the architectural style pipe&filter a connector is responsible for transmitting information from the output of a filter to the input of another one. In this case, the connector does not route messages but transports information flows. Besides, specializations of this style characterize and restrict attributes of the connector like its storage capacity (bounded pipes) or the type of information transported by the connector (typed pipes). In a lay-

ered architectural style, connectors are defined by the protocols determining how layers interact.

In short, the metaclass *Connector* as defined in UML 2.0 does not seem to represent the concept of connector assumed by the software architecture community. As we say in section 5.2.5, the metaclass *Connector* can not represent a C3 connector. From our point of view, it would be necessary to define a new metaclass in UML 2.0 to characterize an architectural connector. This new metaclass should be added to the package *InfrastructureLibrary::Core*. Of course, extending the infrastructure will imply extending also the superstructure with the purpose of defining user level constructors. The same would apply to the concept of role linked to a connector. We are studying the correct way to do these extensions.

6 UML 1.4 vs. UML 2.0 to Describe the Static View of the C3 Architectural Style

In this section we are not going to do a comparison between UML 1.4 and UML 2.0 since other works already treat this problem [13]. The content of this section is limited to present the differences between the results obtained when UML 1.4 is used to describe the static view of the C3 architectural style (Section 4) and the results obtained when UML 2.0 is used to the same goal (Section 5).

The most important difference between the two approaches is that to use UML 1.4 it was necessary to realise a heavyweight extension to the metamodel. The reasons are that there were not constructors which could represent the semantics of the architectural elements of C3 and that we have dismissed the stereotyping mechanism because this mechanism is not able to characterise correctly those architectural elements. Thus we added to the metamodel the following metaclasses: *Architecture*, *Component*, *Connector*, *Filter*, *Port*, *Role* and *InterfaceElement,* and the new enumerated types *Direction* and *Domain*. On the contrary some metaclasses, like *Constraint*, *Parameter* and *Attribute,* and the predefined types *Boolean* and *ProcedureExpresion,* were reused. Nevertheless, a heavyweight extension to the metamodel implies that the tools processing UML 1.4 are not able to process this new extension.

With UML 2.0 the problem is easier to solve because UML 2.0 has incorporated metaclasses to represent architectural elements. The static view of the C3 architectural style has been able to be represented completely using the existing metaclasses and stereotypes over those metaclasses. As we mentioned in section 5.2.5, the only dark point is the treatment of connectors. Bellow we indicate some differences between the two approaches:

- With UML 2.0 the metaclass *Model* has been stereotyped to represent a C3 architecture. With UML 1.4, that concept was represented by adding the metaclass *Architecture* to the metamodel.
- With UML 2.0 the metaclass *Component* has been stereotyped to represent a component. With UML 1.4, that concept was represented by adding to the metamodel the metaclass *Component* that declared the attribute *isActive*. In

UML 2.0, the metaclass *Component* inherits this attribute from the metaclass *Class*.

- With UML 2.0 the metaclass *Component* has been stereotyped to represent a C3 connector. We saw that this is not the best solution. However, it is better than to add two new metaclasses to the metamodel (*Connector* and *Filter*) as we did with UML 1.4.
- With UML 2.0 the metaclass *Port* has been stereotyped to represent the concepts of port and role of C3. With UML 1.4 we had to add the metaclasses *Port* and *Role* to the metamodel.
- With UML 2.0, to represent the interfaces of a component in C3, the metaclasses *Operation* and *Interface* have been stereotyped. With UML 1.4 we added the metaclass *InterfaceElement* to the metamodel. Furthermore, with UML 2.0 it is easier to distinguish the interfaces provided by a component from the interfaces required by that component because in the metamodel the relationships provided and required are already defined between the metaclasses *Component* and *Interface*.

As a conclusion, UML 2.0 provides more features than UML 1.4 to model aspects of a software architecture; mainly the aspects related to the concepts of component, port and interface. Nevertheless, we think that UML 2.0 has not introduced enough semantic power to model the concept of connector as defined by the software architecture community.

7 Conclusions and Future Work

In this work we have proponed an extension to the metamodels of UML 1.4 and UML 2.0 to describe the static view of the C3 architectural style. As we said in Section 4, it was necessary to add new metaclasses to the UML 1.4 metamodel to represent most of the C3 architectural concepts. On the contrary, with UML 2.0 it has been enough to define stereotypes of metaclasses already defined in its metamodel to realise the same function. This shows that the new constructors of UML 2.0, like *Connector* and *Port*, and the new semantics of the metaclass *Component*, allow to describe architectural aspects contrary to happens with UML 1.4. Nevertheless, we found several problems when trying to define the constructor connector in C3 since the metaclass *Connector* of UML 2.0 has different semantics. In fact, in Section 5.2.5 we illustrated some important differences between both concepts that made it impossible for that metaclass to be used as the base class for stereotype *C3Connector*. The core problem is that the metaclass *Connector* defined in UML 2.0 is not a type of *Classifier*, while *Component* is. In UML 2.0 connectors do not appear to be first class entities. From our point of view a revision of UML 2.0 is necessary in order to support the software connector.

In the short term, we plan to do an analysis of the capabilities of UML 2.0 to represent a dynamic (behavioral) view of architectural style C3. Then, our research will focus on the different architectural styles defined nowadays with the purpose of capturing their similarities and establish a set of metaclasses needed to add in UML 2.0 to be able to represent them. The proposed extensions, derived from that study, will

define a new member in the UML family of languages. This set of extensions could be sent to the RTF (Revision Task Force) corresponding to OMG so that they could be considered in the next language review.

References

1. Abi-Antoun, M. and Medvidovic, N. (1999). Enabling the refinement of a software architecture into a design. In Proc. of The Second International Conference on The Unified Modeling Language. CO, USA: Springer-Verlag.
2. Bass, L., Clements, P. and Kazman, R. (1998). Software architecture in practice. Reading, Massachusetts: Addison-Wesley
3. Björkander, M. and Kobryn, C. (2003). Architecting systems with UML 2.0. IEEE Software, 20, 4, 57-61.
4. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. (2003). Documenting software architectures, views and beyond. Boston, Massachusetts: Addison-Wesley.
5. Egyed, A. and Medvidovic, N. (2001). Consistent architectural refinement and evolution using the Unified Modeling Language. In Proc. of the 1st Workshop on Describing Software Architecture with UML. Toronto, Canada, 83-87.
6. Garlan, D., Allen, R. and Ockerbloom, J. (1994). Exploiting style in architectural design environments. In Proc. of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, 175-188.
7. Garlan, D. and Kompanek, A.J. (2000). Reconciling the needs of architectural description with object-modeling notation. UML 2000 – The Unified Modeling Language: Advancing the Standard. Third International Conference. York, UK: Springer-Verlag.
8. Gomaa, H. and Wijesekera (2001). The role of UML, OCL and ADLs in software architecture. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01. Toronto, Canada.
9. Hilliar, R. (1999). Building blocks for extensibility in the UML. Response to UML 2.0 Request For Information". Available from OMG as ad/99-12-12.
10. Hofmeister, C., Nord, R.L. and Soni, D. (1999). Describing software architecture with UML. In Proc. of the First Working IFIP Conf. on Software Architecture. San Antonio, TX: IEEE.
11. IEEE (2000). IEEE Recommended practice for architectural description of software-intensive systems.
12. Kandé, M. M. and Strohmeier, A. (2000). Towards a UML profile for software architecture descriptions. UML 2000 – The Unified Modeling Language: Advancing the Standard. York, UK: Springer-Verlag.
13. Kramler, G. (2003). Overview of UML 2.0 abstract syntax. Available from http://www.big.tuwien.ac.at/staff/kramler/uml/uml2-superstructure-overview.html.
14. Lüer, C. and Rosenblum, D.S. (2001). UML component diagrams and software architecture- experiences from the WREN project. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01. Toronto, Canada.
15. Medvidovic, N. (1999). Architecture-based specification-time software evolution. (Doctoral Dissertation, University of California, Irvine, 1999).
16. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F. and Robbins, J.E. (2002). Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology, 11 (1), 2-57.

17. Mehta, N.R., Medvidovic, N. and Phadke, S. (2000). Towards a taxonomy of software connectors. In Proc. of ICSE'00, ACM. Limerick, Ireland, 178-187.
18. OMG (2001). Unified Modeling Language specification (version 1.4).
19. OMG (2003a). Unified Modeling Language (UML) Specification: Infrastructure, version 2.0 (ptc/03-09-15). http://www.omg.org/uml.
20. OMG (2003b). Unified Modeling Language: Superstructure, version 2.0 (ptc/03-08-02). http://www.omg.org/uml.
21. Pérez-Martínez, J.E. (2003). Heavyweight extensions to the UML metamodel to describe the C3 architectural style. ACM SIGSOFT Software Engineering Notes, Vol. 28 (3).
22. Rausch, A. (2001). Towards a software architecture specification language based on UML and OCL. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01, Toronto, Canada.
23. Riva, C., Xu, J. and Maccari, A. (2001). Architecting and reverse architecting in UML. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01, Toronto, Canada.
24. Rumpe, B., Schoenmakers, M., Radermacher, A. and Schürr, A. (1999). UML + ROOM as a standard ADL? In Proc. of Fifth International Conference on Engineering of Complex Computer System, Las Vegas, Nevada.
25. Selic, B. (2001). On modeling architectural structures with UML. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01. Toronto, Canada.
26. Shaw, M. (1994). Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Tech. Rep. CMU-CS-94-107, Pittsburgh, PA: Carnegie Mellon University, School of Computer Science and Software Engineering Institute.
27. Shaw, M., DeLine, R. and Zelesnik, G. (1996). Abstractions and implementations for architectural connections. In Proc. of 3rd International Conference on Configurable Distributed Systems, Annapolis, Maryland.
28. Shaw, M. and Garlan, D. (1996). Software architecture. Perspectives on an emerging discipline. N.J., USA: Prentice-Hall.
29. Störrle, H. (2001). Turning UML-subsystems into architectural units. In Proc. of the Workshop on Describing Software Architecture with UML, ICSE'01, Toronto, Canada.